TD : trading client implementation

Technical objective

Students learn how to use simple asynchronous rpc to interact with a remote real time process.
They have to follow the technical specification in order to be fully compliant with the required message sequence.

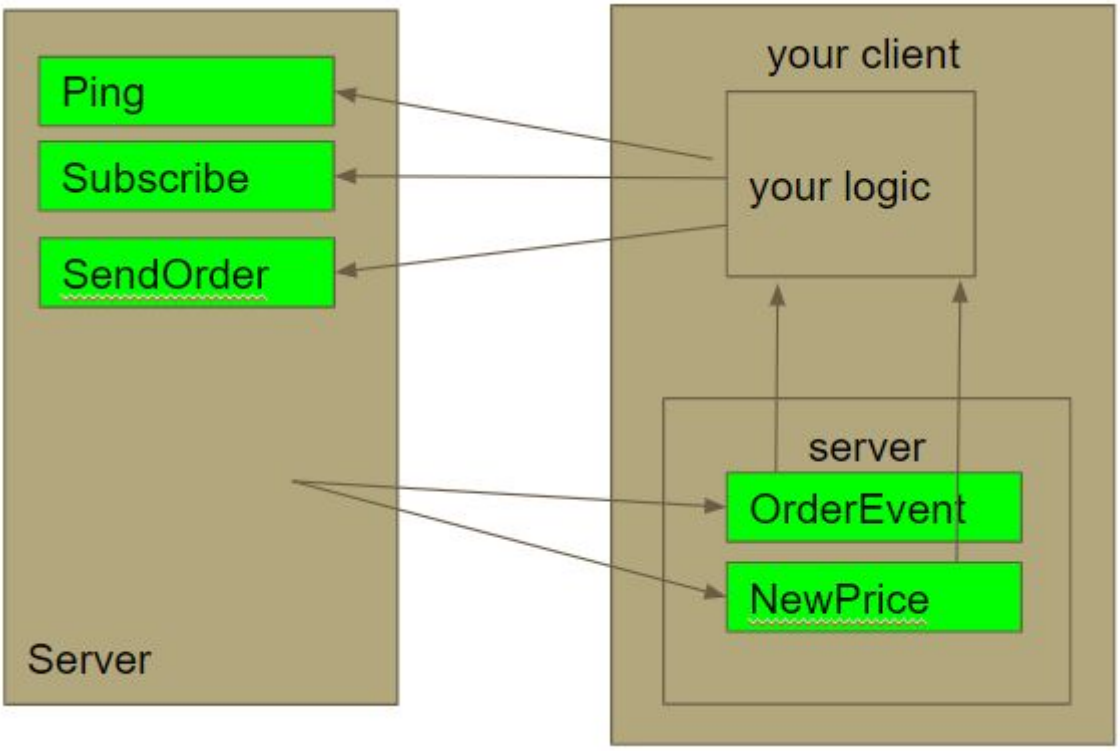
Grpc and protobuf will be used as technical third-party libraries.
Documentation can be found here : https://grpc.io/docs/
Students can find help here : https://stackoverflow.com/ (asking question or just using the QA database)

Functional objective

Students build a simple trading system : they handle the subscription step, detect the trading opportunity and try to send orders to generate profitable deals.

They have to follow the negotiation workflow and observe the impact of latency on their system performance.

The “efrei exchange server”



This server process simulates a market maked product quoted on 2 different exchanges.
It publishes continuously a moving bid/ask with their quantities.
The product is referenced as “0” on first exchange and as “1” on second exchange.

efrei_exchange_server.proto

Through file efrei_exchange_server.proto, exchange server exposes several methods on its ExchangeService endpoint :

rpc PingSrv(Void)returns (Void){}

Clients call this method to test is the server is up and running

rpc Subscribe(SubscribeArgs) returns (SubscribeResponse) {}

Clients call this method to receive the prices on their server. The client server (see below) is started on endpoint provided in the SubscribeArgs object sent to the server.

rpc SendOrder(SendOrderArgs)returns (SendOrderResponse) {}

Clients call this method to send an IOC+FOK order.
IOC behaviour : order is immediately executed, or, if not, cancelled
FOK behaviour : order is executed is the indicated quantity can be found on the market, else cancelled

Through file efrei_exchange_server.proto, client exposes a server able to handle several methods on its ExchangeClient endpoint used by the exchange server to interact :

rpc NewPrice(NewPriceArgs) returns (Void) {}

Called by the exchange server when a new price is available.

rpc OrderEvent(OrderEventArg)returns (Void) {}

Called by the exchange server when a event on a client order is available

rpc Ping(Void)returns (Void){}

Called by the exchange server on subscription to test is the client has been started its ExchangeClient server.

Steps

Choose a langage

Recommanded languages : c++, c#, python, go, java

Install Grpc and protobuf dependencies

1. Install Grpc library version 1.4.1 : this library is used at runtime for server and remote client.
2. Install Google.Protobuf library version 3.0 : this library at runtime for de/serialization (under the hood). The code generation is done as a prerequisite step
3. Install Grpc.Tools : this lib provides the protoc.exe code generator.
4. Verify you got the protoc plugin for your language. If not : find it and build it from sources

Generate the code

For c#

https://grpc.io/docs/quickstart/csharp.html

protoc.exe --csharp_out=/path/to/wanted/folder --grpc_out=/path/to/wanted/folder
--plugin=protoc-gen-grpc=/path/to/plugin.exe efrei_exchange_server.proto

For python

https://grpc.io/docs/quickstart/python.html

python -m efrei_exchange_server.proto --python_out=/path/to/wanted/folder --grpc_python_out=.
efrei_exchange_server.proto

Add the generated files to your project.

Starts the server

In a shell execute :

efrei_exchange_server.exe ip:port

Exemple : efrei_exchange_server.exe 192.168.1.11:10000

Ip : your ip can be found using the ipconfig command in a cmd shell
Port : use any port in range 1024 < port < 65535, for example 10000

Creates a console application to PingSrv

1. Create a console application with your language
2. Add the generated files
3. In the main
    a. creates a channel to the ip:port of the server
    b. Creates a client to the ExchangeEngine
    c. Invoke PingSrv method

⇒ The server should print a “PING” in the console

Creates the ExchangeClient server

1. Create a class inheriting of Efrei.ExchangeServer.**ExchangeClient**.Service class
2. Overrides the 3 methods (NewPrice, OrderEvent, Ping). For now, just print a message on the console
3. In the main, after the ping step
    a. start the server
    b. invoke “Subscribe” passing the endpoint on which the client server started
    c. Check no error occurred
    d. Print the client_id you received
4. Test
    a. launch exchange server
    b. launch your program : you should see something like (depending of your messages)

PING called !
SUBSCRIBE called ! client_id returned : 1351413213
NEWPRICE !
NEWPRICE !
NEWPRICE !
NEWPRICE !
...

Detect opportunities

Modify NewPrice method : print a message when book are crossed (the best buyer/seller of a book is strictly greater/lesser than the best seller/buyer of the other book).

A part of the group can stay on this version of your program and try to “learn” the behaviour of the price movements.

Send an order

Now the client server is started, you have to send orders.

Success order

Invoke SendOrder with a SendOrderArgs arguments built as following:

instrument_id=0,
price=9999999,
qty=1
client_id=(the client_id you received after Subscribe call)

Check carefully the returned SendOrderResponse.
Observe the argument sent to OrderEvent method of your server. Explain the execution price.

Error orders

Invoke SendOrder with the following SendOrderArgs arguments. These will provoke an error which can be found in the return value of the SendOrderArgs call :

{ instrument_id=0, price=9999999, qty=1 client_id=123456} ⇒ explain the technical error received
{ instrument_id=-1, price=9999999, qty=1 client_id=your_client_id} ⇒ explain the technical error received
{ instrument_id=10, price=9999999, qty=1 client_id=your_client_id} ⇒ explain the technical error received
{ instrument_id=0, price=9999999, qty=0 client_id=your_client_id} ⇒ explain the technical error received

Accepted orders but not executed

Now send technically valid orders but functionally not-executable orders

{ instrument_id=0, price=1, qty=1 client_id=your_client_id} ⇒ explain the order status received
{ instrument_id=0, price=1, qty=-99999 client_id=your_client_id} ⇒ explain the order status received

Well done

You reach this step ? Well done !

Now, try to detect opportunity in order to make pnl !

Nothing has to be automatic ...
You can observe the market price movement …
You can have several programs running at the same time …

!!! I WANT POSITIVE PNL GENERATED ON THE EXCHANGE SIDE !!!

Appendix

Keep a portfolio updated

The .proto file contains 2 objects very usefull to follow up portfolio state : PortfolioSingleProduct and Portfolio. When receiving a Deal via the OrderEvent callback, the portfolio of the instrument can be updated using the following method (sorry, c++ but one person in the group can translate it) :

```
// at the end the “p” variable has been updated
void update_portfolio(exchange_server::PortfolioSingleProduct & p, const exchange_server::Deal & d)
	{
		p.set_executed_qty(p.executed_qty() + abs(d.qty()));
		auto deal_qty_left = d.qty();
		while (deal_qty_left != 0)
		{
			if (p.open_qty()*deal_qty_left >= 0) // increase qty
			{
				p.set_open_qty(p.open_qty() + deal_qty_left);
				auto * newdeal = p.mutable_open_deals()->Add();
				newdeal->set_price(d.price());
```

```
                        newdeal->set_qty(deal_qty_left);
                        return;
                }

                auto nxt_qty = p.mutable_open_deals()->rbegin()->qty();
                auto nxt_prc = p.mutable_open_deals()->rbegin()->price();

                if (abs(nxt_qty) >= abs(deal_qty_left)) // this pending exec has enough qty for the deal
                {
                        if (p.open_qty() + deal_qty_left == 0)
                        {
                                p.set_open_qty(0);
                                p.set_pnl(p.pnl() + deal_qty_left * (nxt_prc - d.price()));
                                p.mutable_open_deals()->RemoveLast();

                                return;
                        }

                        p.set_open_qty(p.open_qty() + deal_qty_left);
                        p.set_pnl(p.pnl() + deal_qty_left * (nxt_prc - d.price()));
                        p.mutable_open_deals()->rbegin()->set_qty(nxt_qty + deal_qty_left);

                }
                else // this pending is gonna be consummed
                {
                        p.set_open_qty(p.open_qty() - nxt_qty);
                        p.set_pnl(p.pnl() - nxt_qty * (nxt_prc - d.price()));
                        p.mutable_open_deals()->RemoveLast();
                        deal_qty_left += nxt_qty;
                }
        }
    }
```

Server visual output explanation

The server outputs continuously the 2 order books :

```
       10006
    -- 10005
       10004
    -- 10003
       10002
       10001
       10000
 --    9999
 --    9998
       9997
```

.

This means :

instrument 0 : bid/ask = 9998/9

instrument 1 : bid/ask = 10003/5

Proto description file

It can be found here

Executable exchange server

It can be found here